Collecting Statistics over Runtime Executions *

Bernd Finkbeiner
Max-Planck-Institut für Informatik, Saarbrücken, Germany
Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken, Germany

Sriram Sankaranarayanan and Henny B. Sipma
Computer Science Department, Stanford University, Stanford, CA 94305, USA

Abstract. We present an extension to linear-time temporal logic (LTL) that combines the temporal specification with the collection of statistical data. By collecting statistics over runtime executions of a program we can answer complex queries, such as "what is the average number of packet transmissions" in a communication protocol, or "how often does a particular process enter the critical section while another process remains waiting" in a mutual exclusion algorithm. To decouple the evaluation strategy of the queries from the definition of the temporal operators, we introduce algebraic alternating automata as an automata-based intermediate representation. Algebraic alternating automata are an extension of alternating automata that produce a value instead of acceptance or rejection for each trace. Based on the translation of the formulas from the query language to algebraic alternating automata, we obtain a simple and efficient query evaluation algorithm. The approach is illustrated with examples and experimental results.

1. Introduction

Runtime verification [13, 14, 23] is an alternative approach to program verification in which individual program traces are checked against a specification. Given a trace of a program execution, we report *success* if the trace satisfies the program specification, and *failure* if a fault is detected. Often, however, it is more helpful to watch indicators of an impending failure, such as the number of packet retransmissions in a network, than to wait for an actual violation of the specification.

In this paper, we present an extension to linear-time temporal logic that combines the temporal specification with the collection of statistical data. Instead of *checking* properties like "there are only finitely many transmissions for each packet" (which is vacuously true over finite traces) we *evaluate* queries like "what is the average number of packet transmissions," or "what is the throughput," which provide a good picture of the current network status.

Statistical measures on traces can be useful in many different settings. As program specifications, they can express important additional

* This research was supported in part by NSF(ITR) grant CCR-01-21403, by NSF grant CCR-99-00984-001, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.

properties. For example, observing more memory deallocation calls than memory allocations in a C program indicates an error in the program's handling of dynamically allocated memory. In testing, statistical measures can help discriminate important test cases from the less significant cases, by measuring the closeness of each test case to some ideal case specified by the designers. Another application area is performance profiling. Using the extended logic we can, for example, easily determine the execution time for different function calls.

In our framework, queries are constructed from *experiments*, which form basic observations at individual trace positions, and *aggregate statistics,* which combine the results of multiple experiments. This query language is defined in Section 2. We discuss examples from a communication protocol and a mutual exclusion algorithm in Section 3. Next, we develop an automata-theoretic solution for the evaluation of queries. *Algebraic alternating automata* are an extension of alternating automata that produce a value instead of acceptance or rejection for each trace. We introduce algebraic alternating automata in Section 4 and discuss their evaluation over traces. The translation of queries to automata is described in Section 5. Section 6 discusses experimental results from our prototype implementation. Section 7 concludes the discussion and points to some useful extensions.

Related Work

Program profiling has a long history, exemplified in popular tools like *gprof* [11]. However, this research has concentrated mostly on certain specific types of data like running time and memory leaks. Our approach can be used to develop flexible profiling tools that evaluate user-defined temporal queries.

Runtime verification with linear-time temporal logic has received growing attention recently [13, 14, 23]. Examples include the commercial system Temporal Rover [7], a tool that allows the specifications to be embedded in C, C++, Java, Verilog and VHDL programs. Runtime verification algorithms have also been applied in guiding the Java model checker Java PathFinder developed at NASA [12].

Linear-time temporal logic is a widely used formalism for the specification and verification of reactive and concurrent systems [17]. For static analysis, other extensions to quantitative queries have first been studied in the context of real time systems [8, 9]. Recent work along the same lines includes [1, 5]. Our query language can be seen as a generalization of the logic MinMax Ctl [5]. Temporal logics are also the basis for industrial property specification standards such as openvera assertions [21]. Property specifications for important protocols like PCI are now available commercially with built-in support for specifying and collecting certain statistical data.

Alternating automata [6] are a generalization of nondeterministic automata and $\forall$-automata [16]. Because of their succinctness they are an efficient data structure for many problems in specification and verification [25, 24]. The algebraic alternating automata we define in Section 4 are inspired by the *extended alternating automata* of [4]. There, extended alternating automata are used for static *Query Checking*, which determines the set of propositional formulas that satisfy a temporal query over a program. Our general framework for using alternating automata for runtime verification was reported in [10]. In this paper we concretize the general approach by providing a query language and a translation from queries to alternating automata.

2. Specifying Runtime Statistics

2.1. Programs, States and Traces

In our framework, runtime verification consists of posing *queries* about *program traces*. These queries typically contain expressions over program variables. Neither the queries nor their evaluations depend any further on the program's internal structure. Therefore, it is sufficient to formalize our notion of states and traces. For the sake of simplicity, we assume that all variables are global in scope. Informally, a program state s is some valuation to the program variables. Each variable x receives a valuation $s(x)$ of the appropriate sort. We also extend a state to map well-typed expressions over variables to their appropriate values. Thus, given an expression e, and a state s, $s(e)$ denotes the value of the expression e in state s.

Formally, let Σ be a *many-sorted* algebraic signature and P be a *program* with a finite set of variables X. Each variable $x \in X$ is assumed to have a fixed sort τ_x. A *query expression* e is a term in the term algebra $\mathcal{T}(\Sigma, X)$. Given a Σ-algebra $\mathcal{V}$, a $\mathcal{V}$-state of a program P is a map $s : X \mapsto \mathcal{V}$ such that each variable $x \in X$ is assigned a value of sort τ_x in $\mathcal{V}$. We extend a state s to the unique homomorphism $s : \mathcal{T}(\Sigma, X) \mapsto \mathcal{V}$ that extends s. This allows us to evaluate expressions over variables using the information from s. Therefore, if e is an expression, $s(e)$ is defined by this homomorphism.

An expression with sort boolean is called an *assertion*. The *entailment* relation $\vDash$ is defined such that a state satisfies an assertion ψ, written $s \vDash \psi$, if and only if $s(\psi) = true$.

Queries are evaluated over program traces. Formally, a (P-)*trace* σ of length n is a sequence of states $s_0, s_1, \ldots, s_{n-1}$. Queries may return a value or they may fail. For example, a query in our formalism could be: "*The event e must occur at least once in the trace; what is the earliest time of its occurrence?*" If e occurs at least once, then the query succeeds, and the appropriate time is returned. If not, we need to signal failure by returning a special value. Hence, we assume that all sorts contain a special element $\perp$ to indicate the failure of a query.

We classify queries into two levels. *Experiments* yield a value for each position in the trace. *Aggregate statistics* combine the results of experiments from multiple positions by computing a statistical measure for a part of the trace or the full trace. We first describe experiments, and then move on to aggregate statistics.

2.2. Experiments

Experiments express basic observations about the program trace at a particular position in the trace.

Syntax Given a set of program variables V, and a signature Σ, experiments are defined inductively as follows:

- Base case (state expression): $p : \delta$ is an experiment, where p is an assertion over V and δ a well-typed expression over V;

- Inductive case: if ψ_1 and ψ_2 are experiments, so are

$$
\begin{array}{ll}
\psi_1 \ \wedge_g \ \psi_2 & \textit{(Conjunction)} \\
\psi_1 \ \vee_g \ \psi_2 & \textit{(Disjunction)} \\
\neg_c \, \psi_1 & \textit{(Negation)} \\
\bigcirc_f \ \psi_1 & \textit{(Next)} \\
\psi_1 \, \mathcal{U}_f \ \psi_2 & \textit{(Until)}
\end{array}
$$

  where f is a unary function, g is a binary function, and c is a constant in Σ.

Semantics Experiments are interpreted over program traces in a Σ-algebra $\mathcal{V}$. The *value* of an experiment ψ over a trace $\sigma : \ s_0, s_1, s_2, \ldots s_n$ at position $0 \le j \le n$, written $[\psi]_{(\sigma,j)}$ is defined as follows:

For a state expression:

$$
[p : \delta]_{(\sigma,j)} = \begin{cases} s_j(\delta) & \text{if } s_j \models p \\ \perp & \text{otherwise} \end{cases}
$$

that is, if the assertion p holds at position j, the value of the experiment is the value of δ in s_j; if p does not hold the experiment is a failure, and its value is $\perp$.

For the boolean connectives:

- Conjunction:

$$[\psi_1 \wedge_g \psi_2]_{(\sigma,j)} = \begin{cases} g([\psi_1]_{(\sigma,j)}, [\psi_2]_{(\sigma,j)}) & \text{if } [\psi_1]_{(\sigma,j)} \neq \perp \\ & \text{and } [\psi_2]_{(\sigma,j)} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

  that is, the value of the experiment at position j is the result of function g applied to the values of ψ_1 and ψ_2 at position j, provided both ψ_1 and ψ succeed at j. Otherwise the experiment is a failure.

- Disjunction:

$$[\psi_1 \vee_g \psi_2]_{(\sigma,j)} = \begin{cases} g([\psi_1]_{(\sigma,j)}, [\psi_2]_{(\sigma,j)}) & \text{if } [\psi_1]_{(\sigma,j)} \neq \perp \\ & \text{or } [\psi_2]_{(\sigma,j)} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

  that is, the value of the experiment at position j is the result of function g applied to the values of ψ_1 and ψ_2 at position j, provided at least one of ψ_1 and ψ_2 succeeds at j. Otherwise the experiment is a failure. It is assumed that g is defined when one of the arguments is $\perp$.

- Negation:

$$[\neg_c \psi_1]_{(\sigma,j)} = \begin{cases} c & \text{if } [\psi_1]_{(\sigma,j)} = \perp \\ \perp & \text{otherwise} \end{cases}$$

  that is, the experiment has the value of c in $\mathcal{V}$ if experiment ψ_1 is a failure (returns $\perp$) at j; the experiment is a failure otherwise.

For the temporal operators:

- Next:

$$[\bigcirc_f \psi_1]_{(\sigma,j)} = \begin{cases} f([\psi_1]_{(\sigma,j+1)}) & \text{if } [\psi_1]_{(\sigma,j+1)} \neq \perp \\ & \text{and } j \neq n \\ \perp & \text{otherwise} \end{cases}$$

that is, the value of the experiment at position j is the result of applying function f to the result of the experiment ψ_1 at position $j+1$, provided that experiment succeeded. Otherwise the experiment is a failure.

– Until:

$$[\psi_1 \,\mathcal{U}_f\, \psi_2]_{(\sigma,j)} \;=\; \begin{cases} f([\psi_2]_{(\sigma,k)}) & \text{where } k \text{ is the least } k, \\ & j \leq k \leq n, \text{ such that} \\ & [\psi_2]_{(\sigma,k)} \neq \perp \text{ and} \\ & [\psi_1]_{(\sigma,i)} \neq \perp \\ & \text{for every } i,\ j \leq i < k, \text{ or} \\ & \\ \perp & \text{if no such } k \text{ exists} \end{cases}$$

that is, the value of the experiment at position j is the result of applying the function f to the value of ψ_2 at the earliest position where ψ_2 succeeds, provided ψ_1 succeeds continuously up to that point. Otherwise the experiment is a failure.

EXAMPLE 1. *Consider the trace*

$$\sigma : \langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle, \langle 5,3\rangle, \langle 4,3\rangle$$

where each state $\langle x,y\rangle$ gives the values of two integer variables x and y. The following are examples of simple experiments:

– *The experiment*

$$[x < y : x]_{(\sigma,0)}$$

expressing: "the value of x in the first state of σ if $x < y$ holds", has the value $\perp$, since the first state does not satisfy $x < y$.

– *The experiment*

$$[x \leq y : \langle x,y\rangle]_{(\sigma,0)}$$

expressing: "the tuple $\langle x,y\rangle$ in the first state of σ if $x \leq y$", has the value $\langle 1,1\rangle$, because $x \leq y$ holds in the first state and $\langle 1,1\rangle$ is the value of $\langle x,y\rangle$ in the first state.

– *The experiment*

$$[\ (x \leq y : x)\ \wedge_+\ ((x \leq y : true)\ \mathcal{U}_{id}\ (y = x+2 : y))\ ]_{(\sigma,0)}$$

has the value 4, the sum of the value of x in the first state and the value of y in the third state, as $\langle 1,3\rangle$ is the first state such that $y = x+2$ is true and $x \leq y$ in the first two states.

Remark We can view linear-time temporal formulas with their usual interpretation over traces as a special case of statistical experiments. We translate an assertion p to the state expression $p : true$; for the boolean connectives, we associate the function $\mathrm{T}(x, y) = true$ with conjunction $\wedge_{\mathrm{T}}$ and disjunction $\vee_{\mathrm{T}}$, and the constant $true$ with negation $\neg_{true}$; for the temporal operators, the identity function $id(x) = x$ is associated with Next $\bigcirc_{id}$ and Until $\mathcal{U}_{id}$.

By this translation we obtain a statistical experiment that has value $true$ if the temporal formula is satisfied and that fails otherwise. Such experiments occur often as part of complex queries. In the remainder of the paper we will use temporal formulas directly as subformulas, omitting the constant $true$ and the functions T and id.

2.3. Aggregate Statistics

Aggregate statistics combine the outcome of experiments at multiple positions by computing a statistical measure for a part of the trace or the full trace. Examples of aggregate statistics are the minimum or maximum value of all successful experiments on a trace, or the sum of all outcomes, or just a count of all successful experiments. We assume that these aggregate statistics can be computed in an incremental fashion and that the evaluation order, forward or backward, does not affect the final value.

Syntax An *aggregate expression* is defined inductively as follows.

- Base case (experiment): An experiment is an aggregate expression.
- Inductive case: if ψ_1 and ψ_2 are aggregate expressions and φ is an experiment with sort boolean, then the following are aggregate expressions:

$\psi_1 \wedge_g \psi_2$	*(Conjunction)*
$\psi_1 \vee_g \psi_2$	*(Disjunction)*
$\mathcal{C}_\alpha\, \psi_1$	*(Unconditional Collection)*
$\psi_1\, \mathcal{I}_\alpha\, \varphi$	*(Interval Collection)*

  where g is a binary function in Σ and α is an incrementally computable statistic over finite sequences.

An incrementally computable statistic is a function α over traces $\sigma : s_0, s_1, \ldots, s_n$, such that there exists a binary function f_α such that

$$\alpha(\sigma) = f_\alpha(\ldots(f_\alpha(f_\alpha(\perp, s_n), s_{n-1}), \ldots), s_0).$$

Examples of incrementally computable statistics are *minimum*, which returns the minimum non-$\perp$ value in the trace, *maximum*, which returns the maximum non-$\perp$ value in the trace, *sum*, which returns the sum of all non-$\perp$ values in the trace, and *count*, which returns the number of non-$\perp$ values in the trace. These functions can be computed incrementally by the following binary functions:

$$\begin{array}{lcl} f_{min}(x,y) & = & \text{if } x < y \text{ then } x \text{ else } y, \\ f_{max}(x,y) & = & \text{if } x < y \text{ then } y \text{ else } x, \\ f_{sum}(x,y) & = & x + y, \\ f_{count}(x,y) & = & x + 1, \end{array}$$

with

$$f_\alpha(x,\perp) \;=\; f_\alpha(\perp,x) \;=\; x$$

for $\alpha \in \{min, max, sum\}$ and

$$\begin{array}{lcl} f_{count}(\perp,x) & = & 1, \\ f_{count}(x,\perp) & = & x. \end{array}$$

Semantics Like experiments, aggregate statistics are interpreted over program traces. The value of an aggregate statistic ψ over a trace $\sigma : s_0, s_1, \ldots s_n$ at position $0 \le j \le n$, written $[\psi]_{(\sigma,j)}$ is defined as follows:

- Conjunction and Disjunction are the same as for experiments.
- Unconditional Collection:

  $$[\mathcal{C}_\alpha\, \psi_1]_{(\sigma,j)} \;=\; f_\alpha(\; [\mathcal{C}_\alpha\, \psi_1]_{(\sigma,j+1)} \;,\; [\psi_1]_{(\sigma,j)} \;)$$

  with $[\mathcal{C}_\alpha\, \psi_1]_{(\sigma,n+1)} = \perp$,

- Interval Collection:

  $$[\psi_1\, \mathcal{I}_\alpha\, \varphi]_{(\sigma,j)} \;=\; \begin{cases} f_\alpha(\; [\psi_1\, \mathcal{I}_\alpha\, \varphi]_{(\sigma,j+1)} \;,\; [\psi_1]_{(\sigma,j)} \;) & \text{if } s_j \vDash \varphi \\ \perp & \text{otherwise} \end{cases}$$

  with $[\psi_1 \mathcal{I}_\alpha\, \varphi]_{(\sigma,n+1)} = \perp$,

where f_α is a binary function that incrementally computes the function α.

EXAMPLE 2. *Consider the trace*

$$\sigma : \langle 1,1,2\rangle, \langle 1,2,2\rangle, \langle 1,3,1\rangle, \langle 2,3,1\rangle, \langle 5,3,1\rangle, \langle 5,3,2\rangle$$

where each triple $\langle x, y, z\rangle$ *identifies the values of three integer variables* x*,* y *and* z*. To illustrate the aggregate statistics, we show how some questions about this trace can be expressed as aggregate expressions. For each expression we show its evaluation in terms of the application of the statistic to the values of the experiments involved in the expression.*

– *What is the number of positions in which the value of* x *is the same as the value of* y*?*

$$[\, \mathcal{C}_{count}\, (x = y)\,]_{(\sigma,0)} \quad = \quad count\ (T, \bot, \bot, \bot, \bot, \bot) \quad = \quad 1$$

– *What is the minimum value of* $x + y$ *in the trace?*

$$[\, \mathcal{C}_{min}\, (true : x + y)\,]_{(\sigma,0)} \quad = \quad min\ (2, 3, 4, 5, 8, 8) \quad = \quad 2$$

– *What is the average value of* $x + y$*?*
We define the average of a quantity as the quotient of its sum and its count, i.e.,

$$\mathcal{C}_{avg}\, \varphi \quad = \quad \mathcal{C}_{sum}\, \varphi \wedge_{\div} \mathcal{C}_{count}\, \varphi \ .$$

Then the average value of $x + y$ *can be written as*

$$[\, \mathcal{C}_{avg}\, (true : x + y)\,]_{(\sigma,0)} \quad = \quad 30 \div 6 \quad = \quad 5.$$

– *What is the maximum sum of the values of* x *in intervals where* $z = 2$*?*

$$[\, \mathcal{C}_{max}\, (\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\,]_{(\sigma,0)}$$

This expression shows a nesting of aggregate expressions. We will first evaluate the inner expression at all positions in the trace:

$$\begin{aligned}
[\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,0)} &= sum(1, 1) = 2\\
[\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,1)} &= sum(1) = 1\\
[\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,2)} &= \bot\\
[\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,3)} &= \bot\\
[\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,4)} &= \bot\\
[\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,5)} &= sum(5) = 5
\end{aligned}$$

resulting in

$$\mathcal{C}_{max}\ (\ (true : x)\ \mathcal{I}_{sum}\ (z = 2)\ )\ ]_{(\sigma,0)} = max\ (2, 1, \bot, \bot, \bot, 5) = 5.$$

$$
\begin{array}{c}
\textbf{local} \quad x : \textbf{integer where } r = 1 \\
\left[\begin{array}{l}
\ell_0\text{:}\ \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\text{:}\ \textbf{noncritical} \\
\ell_2\text{:}\ \textbf{request}\ r \\
\ell_3\text{:}\ \textbf{critical} \\
\ell_4\text{:}\ \textbf{release}\ r
\end{array}\right]
\end{array}\right]
\quad || \quad
\left[\begin{array}{l}
m_0\text{:}\ \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_1\text{:}\ \textbf{noncritical} \\
m_2\text{:}\ \textbf{request}\ r \\
m_3\text{:}\ \textbf{critical} \\
m_4\text{:}\ \textbf{release}\ r
\end{array}\right]
\end{array}\right] \\
\quad -P_1- \qquad\qquad\qquad\qquad -P_2-
\end{array}
$$

Figure 1. Program MUX-SEM (mutual exclusion by semaphores)

3. Examples

To illustrate the collection of statistics of running programs, we present a mutual exclusion and a communication protocol and some examples of relevant statistics for these programs. The programs are written in the Simple Programming Language (SPL) of [17], which is a Pascal-like language with constructs for concurrency. Statements are labeled to allow explicit reference to control locations.

3.1. Mutual Exclusion

Figure 1 shows an SPL program that ensures mutually exclusive access to the critical section of two processes by means of a semaphore [17]. The **request** statement is enabled only if r is positive, and when executed, it decrements r by 1. The **release** statement increments r by 1. The following are some example queries on traces of the mutual exclusion algorithm.

- *Semaphore values:* In a correct implementation, the maximum value of r should not exceed 1. The expression

  $$\mathcal{C}_{max}\ (true : r)$$

  can be used to monitor whether this is indeed the case.

- *Mutual exclusion:* The expression

  $$\mathcal{C}_{max}\ (at\_\ell_3 : 1\ \vee_+\ at\_m_3 : 1)$$

  records the maximum number of processes present in the critical section at any one time. The predicate at_ℓ_3 is *true* when process P_1 is in location ℓ_3; similarly, at_m_3 is *true* when process P_2 is in location m_3. If the value of this expression exceeds 1, mutual exclusion is violated.

– *Bias:* The expression

$$\mathcal{C}_{count}(at\_\ell_3 \wedge \bigcirc \neg at\_\ell_3) \ \wedge_{\div}\ \mathcal{C}_{count}(at\_m_3 \wedge \bigcirc \neg at\_m_3)$$

returns the ratio of the number of visits by P_1 to the critical section to the number of visits of P_2 to the critical section.

– *Overtaking:* Program MUX-SEM does not put a bound on how often one process can enter the critical section while the other process is waiting to enter. In practice, one may want to monitor the number of times a process is overtaken. The expression

$$\mathcal{C}_{max}\ ((\neg at\_m_3 \wedge\ \bigcirc\ at\_m_3)\ \mathcal{I}_{count}\ at\_\ell_2)$$

records the maximum number of times P_2 visits the critical section during any period where P_1 idles at ℓ_2.

3.2. Communication Protocol

Figure 2 shows an SPL implementation (adapted from [18]) of the Alternating Bit Protocol, a communication protocol that guarantees data delivery to the receiver across a lossy channel, first proposed in [2]. Two processes, a *sender* and a *receiver* execute in parallel. The sender sends data items via the asynchronous data channel *dchan*; each data item is accompanied by a boolean value *seq* (the alternating bit). It then waits for the receiver to send an acknowledgement, consisting of one bit, on the asynchronous acknowledgement channel *achan*, or it times out (we assume that statement ℓ_4 is taken a fixed amount of time after it becomes enabled). If an ack was received and its value is equal to the seq bit, the sender assumes the data was received and it moves on to the next data item, simultaneously flipping the value of seq. If no ack was received, or its value was not equal to seq, the same data item is sent again. The receiver retrieves the data items from *dchan*. If the accompanying seq bit is equal to its local ack bit, it accepts the data by moving its pointer to the next data item, and flips its ack bit. We assume that both *achan* and *dchan* may lose items, but do not corrupt or reorder items. The following are some example queries on traces of this protocol.

– *Throughput:* The total number of data items successfully sent, can be expressed by

$$\mathcal{C}_{count}(\ at\_\ell_6 \wedge \bigcirc \neg at\_\ell_6).$$

$$
\begin{array}{l}
\quad\textbf{local}\ \ dchan\ :\textbf{channel}\,[1..]\ \textbf{of}\ (\textbf{integer}, \textbf{boolean}) \\
\quad\textbf{local}\ \ achan\ :\textbf{channel}\,[1..]\ \textbf{of boolean} \\
\quad Sender :: \left[\begin{array}{l}
\textbf{local}\ \ data\ :\textbf{array}\,[1..]\ \textbf{of integer} \\
\textbf{local}\ \ i\ :\textbf{integer where}\ i = 1 \\
\textbf{local}\ \ seq, ack\ :\textbf{boolean where}\ seq = \text{TRUE} \\
\textbf{local}\ \ timeout\ :\textbf{boolean where}\ \neg timeout \\
\\
\ell_0\!:\ \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
\ell_1\!:\ dchan \Leftarrow (data[i], seq) \\
\ell_2\!:\ \left[\begin{array}{l}
\ell_3\!:\ achan \Rightarrow ack \\
\ \textbf{or} \\
\ell_4\!:\ timeout := \text{TRUE}
\end{array}\right] \\
\ell_5\!:\ \textbf{if}\ \neg timeout \wedge ack = seq \\
\quad \ell_6\!:\ (i, seq) := (i+1, \neg seq) \\
\ell_7\!:\ timeout := \text{FALSE}
\end{array}\right]
\end{array}\right] \\
|| \\
\quad Receiver :: \left[\begin{array}{l}
\textbf{local}\ \ recvd\ :\textbf{array}\,[1..]\ \textbf{of integer} \\
\textbf{local}\ \ j\ :\textbf{integer where}\ j = 1 \\
\textbf{local}\ \ seq, ack\ :\textbf{boolean where}\ ack = \text{TRUE} \\
\\
m_0\!:\ \textbf{loop forever do} \\
\quad \left[\begin{array}{l}
m_1\!:\ dchan \Rightarrow (recvd[j], seq) \\
m_2\!:\ \textbf{if}\ seq = ack\ \textbf{then} \\
\quad m_3\!:\ (j, ack) := (j+1, \neg ack) \\
m_4\!:\ achan \Leftarrow seq
\end{array}\right]
\end{array}\right]
\end{array}
$$

Figure 2. Program ABP: Alternating Bit Protocol

- *Sent vs. received:* The number of items sent by the Sender versus the number of items received by the Receiver is recorded by

$$\mathcal{C}_{count}\ (at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1) \wedge_{\langle .,. \rangle} \mathcal{C}_{count}(at\_m_1 \wedge \bigcirc \neg at\_m_1).$$

- *Maximum transmissions:* The maximum number of transmissions for any one packet is expressed by

$$\mathcal{C}_{max}\ (\ (at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1)\ \mathcal{I}_{count}\ at\_\ell_{\{1\ldots5,7\}}\ ).$$

  The expression counts the number of times statement ℓ_1 is executed in any interval in which control resides at control locations $\ell_1, \ldots, \ell_5$, or ℓ_7, but not at ℓ_6, where the sender moves to the next data item. It then takes the maximum over all intervals.

- *Average transmissions:* The average number of transmissions per packet can be expressed by a similar expression,

  $$\mathcal{C}_{avg}\ (\ at\_\ell_{\{0,6\}} \wedge_{\pi_2} \bigcirc((at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1)\ \mathcal{I}_{count}\ at\_\ell_{\{1\ldots 5,7\}})\ ).$$

  In each position, the interval collection computes the number of transmissions for the current packet. The conjunction with $at\_\ell_{\{0,6\}}$ ensures that we count each packet only once in computing the average, as the value of the conjunction is non-$\perp$ only in the positions where the sender moves from ℓ_0 to ℓ_1 or from ℓ_6 to ℓ_7 and hence starts with a new data item.

4. Evaluating Statistics

Queries are evaluated over traces. To decouple the evaluation strategy from the definition of the temporal operators, we introduce *algebraic alternating automata* as an intermediate representation. Algebraic alternating automata are an extension of alternating automata that produce a value instead of acceptance or rejection for a given trace. We split the evaluation of an expression over a trace into two steps: first, the expression is translated into an equivalent automaton; then, that automaton is evaluated over the trace.

4.1. Alternating Automata

Alternating automata were first introduced in [6]. They provide a concise representation for temporal properties: an LTL formula can be translated into an equivalent alternating automaton that is linear in the length of the formula [20, 26]. In fact, there is a one-to-one relationship between the subformulas of the temporal formula and the states of the automaton, which provides a good intuition for manipulating the automaton in relation to the corresponding formula. Alternating automata have been represented in various ways in the literature. Here we use the definition of [19].

DEFINITION 1 (Alternating Automaton). *An* alternating automaton $\mathcal{A}$ *is defined as follows:*

$\mathcal{A}$	$::=$	$\epsilon_{\mathcal{A}}$	*empty automaton*
	$\mid$	$\langle \nu, \delta, f \rangle$	*single node*
	$\mid$	$\mathcal{A} \wedge \mathcal{A}$	*conjunction of two automata*
	$\mid$	$\mathcal{A} \vee \mathcal{A}$	*disjunction of two automata*

where ν is a state formula, δ is an alternating automaton expressing the next-state relation, and f indicates whether the node is accepting (denoted by $+$) or rejecting (denoted by $-$). We require that the automaton be finite.

4.2. Algebraic Alternating Automata

Algebraic alternating automata are an extension of alternating automata. We distinguish two types of single nodes: *terminal* nodes, which correspond to nodes with next-state relation $\epsilon_{\mathcal{A}}$ in Definition 1, and *transient* nodes, which correspond to single nodes with a next-state relation that is different from the empty automaton. A terminal node is labeled with an assertion and an expression, where the type of the expression determines the type of the automaton. A transient node is labeled with a unary function and has a next-state relation. Conjunction and disjunction of automata is associated with a binary function. Finally, an automaton can be constructed by function application to another automaton. Note that conjunction, disjunction, and function application do not add any new nodes to the automaton.

DEFINITION 2 (Algebraic Alternating Automaton). *Let Σ be a many-sorted algebraic signature and X be a set of* program *variables. An* algebraic alternating automaton *of sort τ is defined as follows:*

$$
\begin{array}{rcll}
\mathcal{A}:\tau & ::= & \langle p,\delta\rangle & \textit{terminal node} \\
 & & & \textit{with assertion } p \textit{ and } \delta:\tau\in\mathcal{T}(\Sigma,X) \\
 & \mid & \langle \mathcal{A}:\tau_1, f\rangle & \textit{transient node} \\
 & \mid & \mathcal{A}:\tau_1 \wedge_g \mathcal{A}:\tau_2 & \textit{conjunction} \\
 & \mid & \mathcal{A}:\tau_1 \vee_g \mathcal{A}:\tau_2 & \textit{disjunction} \\
 & \mid & f(\mathcal{A}:\tau_1) & \textit{function application}
\end{array}
$$

where $f:\tau_1\rightarrow\tau$ and $g:\tau_1\times\tau_2\rightarrow\tau$ are a unary and a binary function, respectively.

EXAMPLE 3. *Figure 3 shows an example of an algebraic alternating automaton over the signature Σ, containing as single sort the natural numbers, a single constant $\perp$ for the undefined value, and the functions π_2, f_{min} and id. Nodes without outgoing edges denote terminal nodes; nodes with an outgoing edge are transient nodes $\langle\mathcal{A}, f\rangle$, with the edge leading to the next-state automaton $\mathcal{A}$. Conjunction of two automata is indicated by an arc connecting the two branches. Thus, the automaton in Figure 3 has three nodes: two terminal nodes (n_0 and n_2) and one transient node (n_1). Automaton $\mathcal{A}_0$ is a conjunction of $\mathcal{A}_1$ and $\mathcal{A}_2$ and automaton $\mathcal{A}_2$ is a disjunction of $\mathcal{A}_3$ and $\mathcal{A}_4$.*

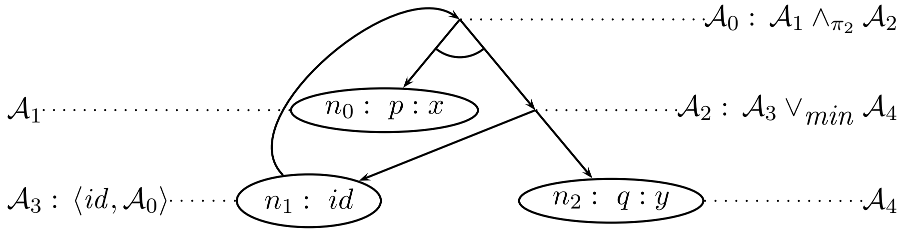

Figure 3. Algebraic Alternating Automaton

DEFINITION 3 (Value). *Given a trace $\sigma : s_0, \ldots, s_n$ and a position j, $0 \leq j \leq n$, the* value *of the algebraic alternating automaton $\mathcal{A}$ at position j, written $[\mathcal{A}]_{(\sigma,j)}$ is defined as follows:*

- *For a terminal node:*

$$[\langle p, \delta \rangle]_{(\sigma,j)} = \begin{cases} s_j(\delta) & \text{if } s_j \vDash p \\ \\ \perp & \text{otherwise} \end{cases}$$

that is, the value is equal to the evaluation of δ at position j in the trace, if the assertion p holds at j.

- *For a transient node:*

$$[\langle \mathcal{A}_1, f \rangle]_{(\sigma,j)} = \begin{cases} f([\mathcal{A}_1]_{(\sigma,j)}) & \text{if } [\mathcal{A}_1]_{(\sigma,j+1)} \neq \perp \\ & \quad \text{and } j \neq n \\ \\ \perp & \text{otherwise} \end{cases}$$

that is, the value is equal to the application of the function f to the value of the next-state automaton $\mathcal{A}_1$, or $\perp$ if j is the last position in the trace, or if the value of the next-state automaton is $\perp$.

- *For a conjunction:*

$$[\mathcal{A}_1 \wedge_g \mathcal{A}_2]_{(\sigma,j)} = \begin{cases} g([\mathcal{A}_1]_{(\sigma,j)}, [\mathcal{A}_2]_{(\sigma,j)}) \\ \qquad \text{if } [\mathcal{A}_1]_{(\sigma,j)} \neq \perp \text{ and } [\mathcal{A}_2]_{(\sigma,j)} \neq \perp \\ \\ \perp \qquad \text{otherwise} \end{cases}$$

that is, the value of a conjunction of automata is equal to the function g applied to the values of the two sub-automata, provided both evaluate to non-$\perp$ values.

– *For a disjunction:*

$$[\mathcal{A}_1 \vee_g \mathcal{A}_2]_{(\sigma,j)} = \begin{cases} g([\mathcal{A}_1]_{(\sigma,j)}, [\mathcal{A}_2]_{(\sigma,j)}) \\ \qquad \text{if } [\mathcal{A}_1]_{(\sigma,j)} \neq \bot \text{ or } [\mathcal{A}_2]_{(\sigma,j)} \neq \bot \\ \\ \bot \qquad \text{otherwise} \end{cases}$$

that is, the value of a disjunction of automata is equal to the function g applied to the values of the two sub-automata, provided at least one of them evaluates to a non-$\bot$ value.

– *For function application:*

$$[f(\mathcal{A}_1)]_{(\sigma,j)} \;=\; f([\mathcal{A}_1]_{(\sigma,j)})$$

that is, the value is equal to the result of applying f to the value of $\mathcal{A}_1$, where $f(\bot)$ may have a non-$\bot$ value and the result of f applied to a non-$\bot$ value may be $\bot$.

EXAMPLE 4. *Consider again the automaton $\mathcal{A}_0$ in Figure 3 and let $\mathcal{V}$ be the Σ-Algebra with carrier set $A = \mathcal{N} \cup \{\bot\}$, where $\mathcal{N}$ is the set of natural numbers, function $\pi_2(x_1, x_2) = x_2$, f_{min} the minimum function over integers that returns the non-$\bot$ value if one of the arguments is $\bot$, and id the identity function. Figure 4 shows the evaluation tree of $\mathcal{A}_0$ over the trace shown on the right of the figure. The trace shows the values of two variables x and y in each position and the satisfaction of two assertions p and q over the program variables.*

Starting at position 6, the end of the trace, values for the terminal nodes n_0 and n_2 are computed directly based on the values of p, q, x, and y at that position. The value of the transient node n_1 at each position is computed based on the result of the value of $\mathcal{A}_0$ in the next position. The value of n_1 at the end of the trace is $\bot$. The value of $\mathcal{A}_0$ over the whole trace is 1, the minimum of the values at nodes n_1 and n_2 at position 0.

4.3. Evaluation on Traces

Evaluation can proceed in the forward direction or in the backward direction. The former strategy traverses the trace from the beginning to the end. The automaton is evaluated recursively, as dictated by the equations in Definition 3. Unfortunately, the complexity of forward evaluation is exponential in the length of the trace. This can be avoided, as pointed out by Rosu and Havelund [22], by traversing the trace backwards.

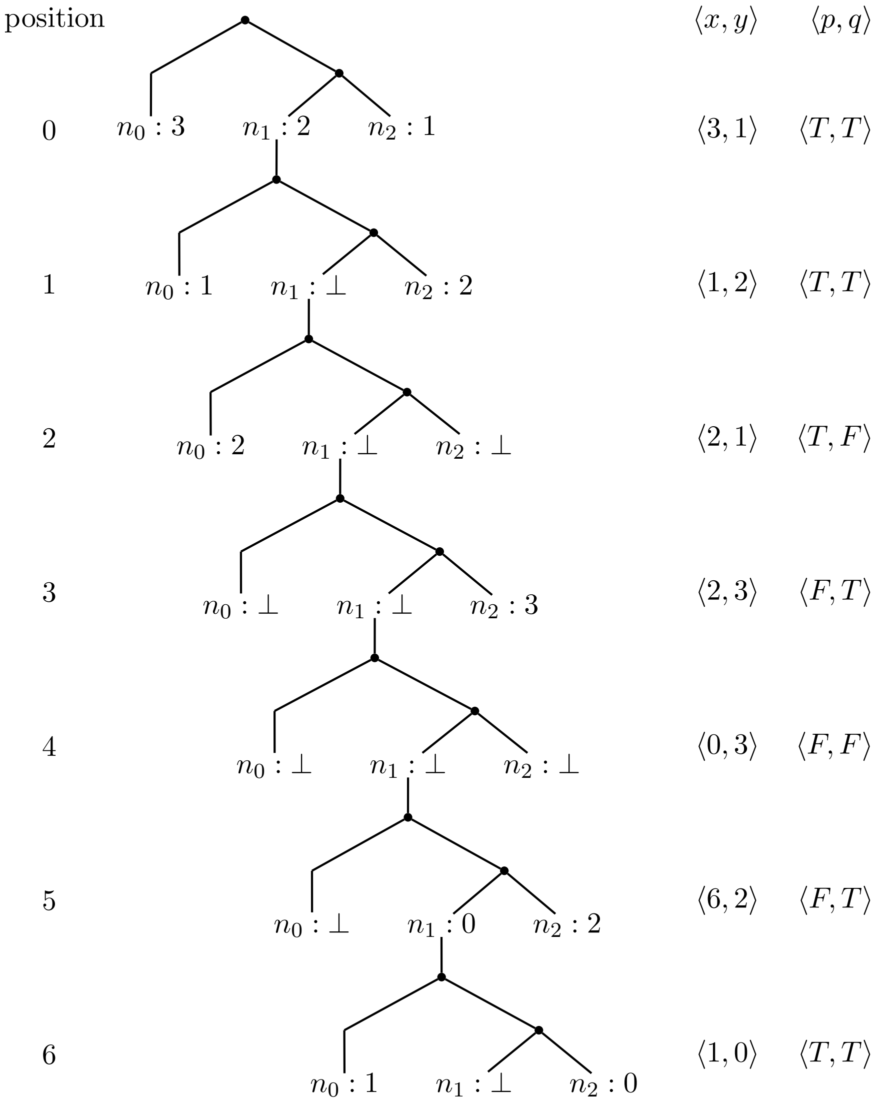

Figure 4. Evaluation tree for $\mathcal{A}_0$ and the trace shown on the right

In the case of backwards traversal, the value of each terminal node is computed for the last state of the trace and all transient nodes are initialized with $\perp$. Then, for each previous state in the trace the new values of the nodes can be computed from the state information at that position and the values of the sub-automata at the previous position. Therefore, it is possible to perform a backwards evaluation while storing the values of all automata at the current and the next positions only.

5. Translating Specifications to Automata

In this section, we describe the translation of formulas of the query language to the algebraic alternating automata described in the previous section. We assume that the algebraic signature Σ contains a binary function f_α for the incremental computation of each aggregate statistic α. In addition, to model the negation operator, we assume that for each constant c in Σ there exists a function $negate_c$ defined as

$$negate_c(v) = \begin{cases} \bot & \text{if } v \neq \bot \\ c & \text{otherwise.} \end{cases}$$

We also assume that each sort has the identity function id.

5.1. Experiments

An experiment ψ is translated into its corresponding algebraic alternating automaton $\mathcal{A}_A(\psi)$ as follows.

For a state expression:

$$\mathcal{A}_A(\psi : e) \quad = \quad \langle \psi, e \rangle$$

the corresponding automaton is a single terminal node.

For the boolean connectives:

- Conjunction of experiments is translated into conjunction of automata:
$$\mathcal{A}_A(\psi_1 \ \wedge_g \ \psi_2) \ = \ \mathcal{A}_A(\psi_1) \ \wedge_g \ \mathcal{A}_A(\psi_2)$$
- Likewise, disjunction of experiments is translated into disjunction of automata:
$$\mathcal{A}_A(\psi_1 \ \vee_g \ \psi_2) \ = \ \mathcal{A}_A(\psi_1) \ \vee_g \ \mathcal{A}_A(\psi_2)$$
- Negation is translated into function application:
$$\mathcal{A}_A(\neg_c \, \psi) \ = \ negate_c(\mathcal{A}_A(\psi))$$

For the temporal operators:

- The Next operator is translated into a transient node:
$$\mathcal{A}_A(\bigcirc_f \, (\psi)) \quad = \quad \langle \mathcal{A}_A(\psi), f \rangle$$

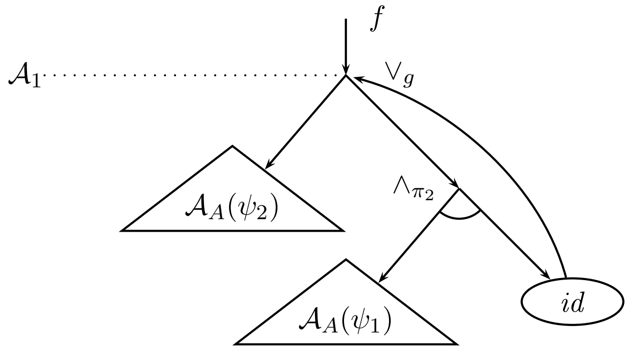

Figure 5. Automaton for $\psi_1\, \mathcal{U}_f\, \psi_2$

- An Until expression is translated into an automaton with a loop as follows:

$$\mathcal{A}_A(\psi_1\, \mathcal{U}_f\, \psi_2) \;=\; f(\, \mathcal{A}_1\,)$$

with

$$\mathcal{A}_1 \;=\; \mathcal{A}_A(\psi_2) \;\vee_g\; (\mathcal{A}_A(\psi_1) \;\wedge_{\pi_2} \langle \mathcal{A}_1, id\rangle)$$

where

$$g(x,y) \;=\; \begin{cases} x & \text{if } x \neq \bot \\ y & \text{otherwise} \end{cases}$$

and $\pi_2(x,y) = y$. The construction is illustrated in Figure 5.

5.2. Aggregate Statistics

Aggregate expressions can be translated to automata in a similar way as experiments. The construction for conjunction and disjunction is identical to those for experiments. The constructions for the unconditional and interval collection are as follows:

- For unconditional collection:

$$\mathcal{A}_A(\mathcal{C}_\alpha(\psi)) \;=\; \mathcal{A}_A(\psi) \;\vee_{f_\alpha}\; \langle \mathcal{A}_A(\mathcal{C}_\alpha(\psi)), id\rangle$$

as illustrated in Figure 6. The transient node labeled with *id* collects the value of the aggregate statistic from the next state; with the disjunction this value is combined with the value of $\mathcal{A}_A(\psi)$ in the current state.

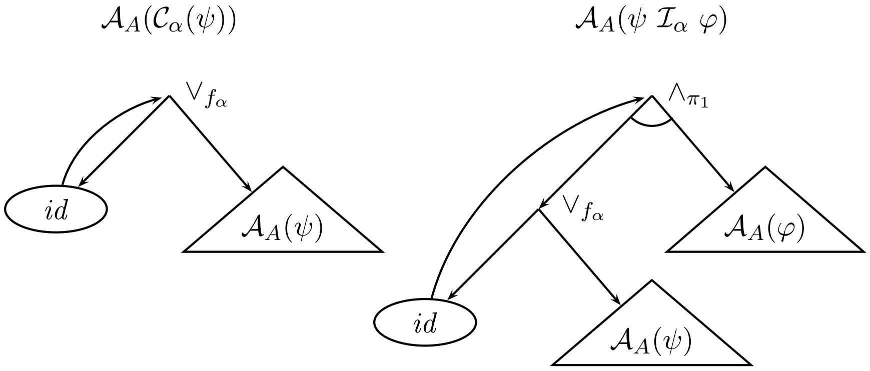

Figure 6. Automata for $\mathcal{C}_\alpha(\psi)$ and $\psi\ \mathcal{I}_\alpha\ \varphi$

- For interval collection:

$$\mathcal{A}_A(\psi\ \mathcal{I}_\alpha\ \varphi) \quad = \quad (\langle\ \mathcal{A}_A(\psi\ \mathcal{I}_\alpha\ \varphi), id\,\rangle \vee_{f_\alpha} \mathcal{A}_A(\psi))\ \wedge_{\pi_1}\ \mathcal{A}_A(\varphi)$$

  as illustrated in Figure 6. The construction is the same as for unconditional collection except that the transmission of the value from the rest of the trace is broken by the conjunction when the value of $\mathcal{A}_A(\psi_2)$ is $\perp$.

EXAMPLE 5. *Figure 7 shows the automaton for the aggregate statistic*

$$\mathcal{C}_{max}(\ (at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1)\ \mathcal{I}_{count}\ at\_\ell_{\{1\ldots5,7\}}\ )$$

expressing the maximum number of transmissions in the communication protocol example from Section 3.2.

6. Experimental Results

The evaluation algorithm from Section 4.3 has been implemented in Java, making use of existing software modules for expression parsing and propositional simplification available in the STeP (Stanford Temporal Prover) system [3]. The formulas described in the mutual exclusion and alternating bit protocol examples of Section 3 were manually translated following the construction from Section 5. Traces were generated by simulating the SPL programs, executing at each position a single step of a randomly chosen process. We measured the running time of the backward evaluation over traces of varying length. The

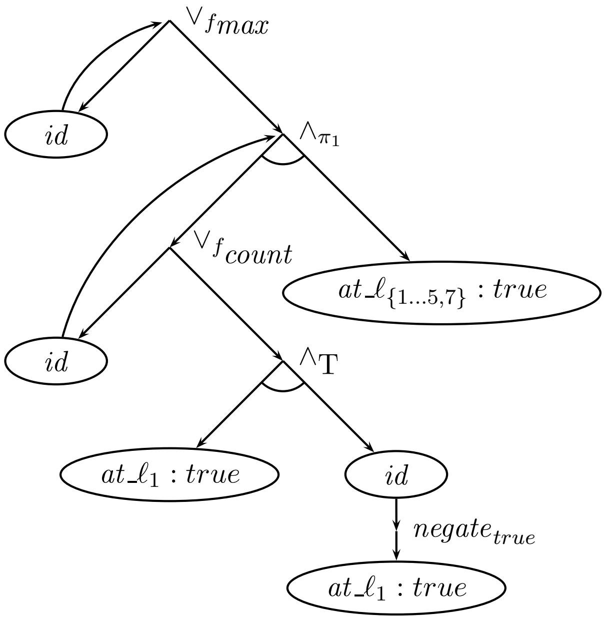

Figure 7. Automaton for $\mathcal{C}_{max}$ ($(at\_\ell_1 \wedge \bigcirc \neg at\_\ell_1)$ $\mathcal{I}_{count}$ $at\_\ell_{\{1...5,7\}}$

results are shown in Figure 8. The times were measured on a 1.7GHz PC, running SuSE Linux v9.0 and Sun JDK 1.5.0.

7. Conclusions & Future Work

We have presented an expressive query language along with a corresponding automata-based representation. Similar to the standard construction of alternating automata from LTL formulas, the translation from our query language to algebraic alternating automata works incrementally by translating subformulas and produces an automaton that is linear in the size of the formula.

It would be very useful to extend our query language to include statistical measures that cannot be computed incrementally, such as the median, quantiles, histograms, modes, and correlations. Computing such measures on a long trace can be expensive in terms of memory, or downright infeasible in terms of time. However, many *randomized* and *approximation* algorithms exist that allow the user to compute good approximations fast, guaranteed to be within a certain accuracy bound with a high probability [15]. The performance of these algorithms can

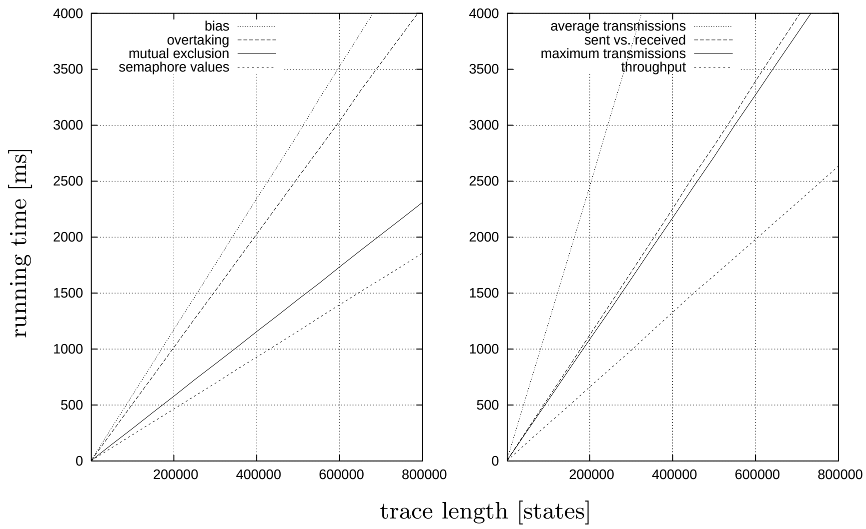

Figure 8. Running times for queries from the mutual exclusion example (left) and the communication protocol example (right).

be traded-off against its time and space complexity. Such algorithms could be used as building blocks to provide a performance guaranteed algorithm for complex statistical measures.

Acknowledgements

We would like to thank the referees for their thorough reading of our submission and their many constructive comments and suggestions. Many thanks to Prof. Zohar Manna and the STeP group for their support and constructive comments.

References

1. Alur, R., S. L. Torre, K. Ettessami and D. Peled, *Parametric temporal logic for model measuring*, in: J. Wiedermann, P. van Emde Boas and M. Nielsen, editors, *ICALP'99, Prague, Czech Republic*, LNCS **1644** (1999), pp. 159–168.
2. Bartlett, K., R. Scantlebury and P. Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Communications of the ACM **12** (1969), pp. 260–261.
3. Bjørner, N. S., A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma and T. E. Uribe, *Verifying temporal properties of reactive systems: A STeP tutorial*, Formal Methods in System Design **16** (2000), pp. 227–270.
4. Bruns, G. and P. Godefroid, *Temporal logic query checking*, in: *Proc. 16th IEEE Symp. Logic in Comp. Sci.* (2001), pp. 409–417.

5. Chakrabarti, P., P. Dasgupta, J. Deka and S. Sankaranarayanan, *Min-max computation tree logic*, Artificial Intelligence **127** (2001), pp. 137–162.
6. Chandra, A. K., D. C. Kozen and L. J. Stockmeyer, *Alternation*, J. ACM **28** (1981), pp. 114–133.
7. Drusinsky, D., *The Temporal Rover and the ATG Rover*, in: K. Havelund, J. Penix and W. Visser, editors, *SPIN Model Checking and Software Verification, 7th Int'l SPIN Workshop*, LNCS **1885** (2000), pp. 323–330.
8. Emerson, A., A. Mok, A. P. Sistla and J. Srinivasan, *Quantitative temporal reasoning*, Real Time Systems **4** (1993), pp. 334–351.
9. Emerson, A. and R. Trefler, *Generalized quantitative temporal reasoning: An automata-theoretic approach*, in: *TAPSOFT: 7th International Joint Conference on Theory and Practice of Software Development*, 1997.
10. Finkbeiner, B. and H. Sipma, *Checking finite traces using alternating automata*, in: K. Havelund and G. Rosu, editors, *Electronic Notes in Theoretical Computer Science*, Electronic Notes in Theoretical Computer Science **55** (2001), pp. 1–17.
11. Graham, S. L., P. B. Kessler and M. K. McKusick, *gprof: a call graph execution profiler*, in: *SIGPLAN Symposium on Compiler Construction*, 1982, pp. 120–126.
12. Havelund, K., *Using runtime analysis to guide model checking of java programs*, in: K. Havelund, J. Penix and W. Visser, editors, *SPIN Model Checking and Software Verification, 7th Int'l SPIN Workshop*, LNCS **1885** (2000), pp. 245–264.
13. Havelund, K. and G. Rosu, editors, "Runtime Verification 2001," Electronic Notes in Theoretical Computer Science **55**, Elsevier Science Publishers, 2001.
14. Havelund, K. and G. Rosu, editors, "Runtime Verification 2002," Electronic Notes in Theoretical Computer Science **70**, Elsevier Science Publishers, 2002.
15. Manku, G. S., Rajagopalan, S. and B. G. Lindsay. *Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets*, in: Proc. ACM SIGMOD, vol. 27(2), pp. 251–262, 1998.
16. Manna, Z. and A. Pnueli, *Specification and verification of concurrent programs by ∀-automata*, in: B. Banieqbal, H. Barringer and A. Pnueli, editors, *Temporal Logic in Specification*, number 398 in LNCS, Springer-Verlag, Berlin, 1987 pp. 124–164, also in *Proc. 14th ACM Symp. Princ. of Prog. Lang.*, Munich, Germany, pp. 1–12, January 1987.
17. Manna, Z. and A. Pnueli, "Temporal Verification of Reactive Systems: Safety," Springer-Verlag, New York, 1995.
18. Manna, Z. and A. Pnueli, "Temporal Verification of Reactive Systems: Progress," Springer-Verlag, New York, 1996, draft manuscript.
19. Manna, Z. and H.B. Sipma, *Alternating the Temporal Picture for Safety*, In: U. Montanari, J. D. Rolim, and E. Welzl (eds.): *Proc. 27th Intl. Colloq. Aut. Lang. Prog.*, Vol. 1853. Geneva, Switzerland, pp. 429–450, Springer-Verlag, 2000.
20. Muller, D.E., A. Saoudi, and P.E. Schupp. *Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time*, In *Proc. 3rd IEEE Symp. Logic in Comp. Sci.*, pages 422–427, 1988.
21. Synopsys inc., "openvera (tm) Assertions," `http://www.open-vera.com`.
22. Rosu, G. and K. Havelund, *Synthesizing dynamic programming algorithms from linear temporal logic formulae*, RIACS Technical Report TR 01-15, 2001.
23. Sokolsky, O. and M. Viswanathan, editors, "Runtime Verification, 2003," Electronic Notes in Theoretical Computer Science **89**, Elsevier Science Publishers, 2003.

24. Vardi, M. Y., *Alternating automata and program verification*, in: J. van Leeuwen, editor, *Computer Science Today. Recent Trends and Developments*, LNCS **1000**, Springer-Verlag, 1995 pp. 471–485.
25. Vardi, M. Y., *An automata-theoretic approach to linear temporal logic*, in: F. Moller and G. Birtwistle, editors, *Logics for Concurrency. Structure versus Automata*, LNCS **1043** (1996), pp. 238–266.
26. Vardi, M. Y., *Alternating automata: Checking truth and validity for temporal logics*, In *Proc. of the* 14^{th} *Intl. Conference on Automated Deduction*, LNCS **1249**, Springer-Verlag, July 1997.